Physically Based Rendering (4th Ed): Chapter 5 Summary

Cameras and Film: Projective Geometry for Programmers

Gemini (For a Developer Audience)

November 29, 2025

Introduction: The Reverse Process

In the real world, cameras capture light. In a ray tracer, cameras **generate** rays.

Chapter 5 is about the math of converting a 2D pixel coordinate (x, y) on your screen into a 3D ray $(Origin, Direction)$ in the world. This is the starting point of every single path in your renderer.

If you know linear algebra (matrices), this chapter is straightforward. If not, we will simplify it to "Coordinate Spaces."

1 Coordinate Systems

Rendering is just moving data between different coordinate systems. You need to know these four spaces by heart:

1. **Object Space**: The local coordinates of a 3D model (e.g., a sphere is centered at 0,0,0).
2. **World Space**: The global scene where all objects live.
3. **Camera Space**: The world seen from the camera's point of view. The camera is at $(0, 0, 0)$, looking down the $+Z$ axis (in PBRT's convention).
4. **Screen Space (Raster)**: The 2D coordinates of the pixels on your image (e.g., pixel 100, 200).

The Pipeline: To generate a ray, we go backwards:

$$\text{Raster} \rightarrow \text{Camera} \rightarrow \text{World}$$

2 The Projective Camera Models

2.1 1. The Pinhole Camera (Perspective)

This is the standard camera. Objects get smaller as they get further away.

The Math (Simplified): Imagine the film (your screen) is placed 1 meter in front of the camera. If you want to generate a ray for a pixel at (x, y) on the film:

- **Origin**: $(0, 0, 0)$ (The camera position).
- **Target**: $(x, y, 1)$ (The point on the film).
- **Direction**: Normalize(Target - Origin).

Field of View (FOV): The FOV determines how "wide" the film is. A wider film captures more of the scene.

$$\tan(\frac{FOV}{2}) = \text{Screen Window Size} \tag{1}$$

2.2 2. The Orthographic Camera

This is used for architectural diagrams or 2D games. Parallel lines remain parallel. Objects do not get smaller with distance.

The Math:

- **Origin**: $(x, y, 0)$ (The ray starts at the pixel's location).
- **Direction**: $(0, 0, 1)$ (Straight forward).

Notice the difference? Perspective rays start at a point and spread out. Orthographic rays start spread out and go parallel.

3 Depth of Field (The Thin Lens Model)

Real cameras are not pinholes. They have lenses. This causes **Depth of Field** (blur).

- Objects at the **Focal Distance** are sharp.
- Objects closer or further are blurry.

How to Simulate Blur? In a pinhole camera, every ray for a pixel starts at the exact same point $(0, 0, 0)$. In a lens camera, the ray can start anywhere on the **Lens Disk**.

The Algorithm:

1. Calculate the point where the perfect pinhole ray hits the **Plane of Focus**. Let's call this P_{focus}.
2. Pick a random point on the lens, P_{lens}. (Remember Chapter 2: Sampling a Disk!).
3. The new ray starts at P_{lens} and goes towards P_{focus}.

```
// 1. Standard Pinhole Ray
Ray ray = GeneratePinholeRay(pixel);
Point3 pFocus = ray.at(focalDistance);

// 2. Sample Lens (Aperture)
Point2 lensSample = SampleConcentricDisk(random1, random2) * lensRadius;
Point3 pLens = Point3(lensSample.x, lensSample.y, 0);

// 3. New Ray
ray.Origin = pLens;
ray.Direction = Normalize(pFocus - pLens);
```

This is a perfect example of Monte Carlo.

- If you take 1 sample, the image looks wrong (noisy/random).
- If you take 100 samples and average them, the intersection of all those rays at P_{focus} remains sharp. The intersection of rays elsewhere spreads out $\rightarrow$ **Blur**.

4 Transformations and Matrices

You don't need to perform matrix multiplication by hand, but you must understand what the matrices *do*.

4.1 The LookAt Matrix

This is the most common helper function. You provide:

- **Eye**: Where the camera is.
- **Look**: What it's looking at.
- **Up**: Which way is "up" (usually Y-axis).

It returns a 4×4 matrix that transforms World Space $\rightarrow$ Camera Space.

4.2 Ray-Object Intersection

Here is a pro-tip from the book. Intersecting a ray with a generic transformed object (e.g., a rotated, scaled ellipsoid) is hard. Intersecting a ray with a unit sphere (radius 1 at origin) is easy.

The Trick: Instead of transforming the *Object* to World Space, we transform the *Ray* to Object Space.

1. Apply the **Inverse** transformation to the Ray.
2. Intersect Ray with Unit Sphere.
3. The hit time t is the same in both spaces.

$$\text{WorldRay} = M \times \text{ObjectRay} \quad \rightarrow \quad \text{ObjectRay} = M^{-1} \times \text{WorldRay} \tag{2}$$

Summary for the Developer

1. **Coordinate Spaces**: Know where your data is. Are you in World Space? Camera Space? Screen Space?
2. **Camera = Ray Generator**: The ‘Camera‘ class has one main method: ‘GenerateRay(pixel)‘.
3. **Depth of Field**: It’s just jittering the ray origin on a disk.
4. **Transform the Ray, Not the Object**: It’s mathematically easier to move the ray into the object’s local coordinate system than to solve math for arbitrary rotated shapes.